1 Virtual Memory

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
  - One major advantage of this scheme is that programs can be larger than physical memory.
  - Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, *separating logical memory as viewed by the user from physical memory.*
- Virtual memory also allows processes to share files easily and to implement shared memory.
- Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly.

1.1 Background

- The instructions being executed must be in physical memory.
- An examination of real programs shows us that, in many cases, the entire program (in memory) is not needed.
  - Programs often have code to handle unusual error conditions (seldom used).
  - Arrays, lists, and tables are often allocated more memory than they actually need.
  - Certain options and features of a program may be used rarely.
- The ability to execute a program that is only partially in memory would offer many benefits:
  - A program would no longer be constrained by the amount of physical memory that is available (simplifying the programming task).
  - Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
  - Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

- **Virtual memory** involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (see Fig. 1).

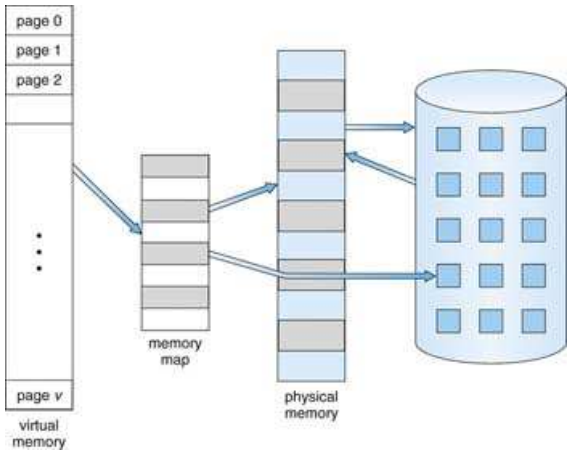

Figure 1: Diagram showing virtual memory that is larger than physical memory.

- The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address-say, address O -and exists in contiguous memory, as shown in Fig. 2.

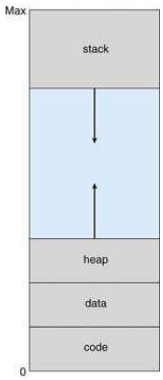

Figure 2: Virtual address space.

- The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.
  - **heap** to grow upward in memory as it is used for dynamic memory allocation
  - **stack** to grow downward in memory through successive function calls
- Virtual address spaces that include holes are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.
- In addition to separating logical memory from physical memory, virtual memory also allows files and memory to be shared by two or more processes through page sharing. This leads to the following benefits:

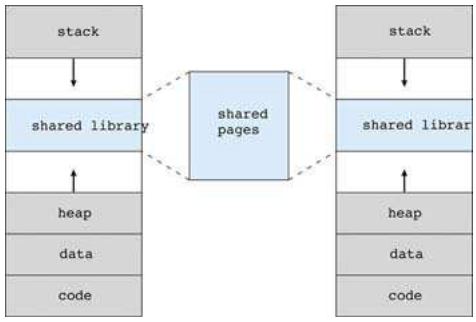

Figure 3: Shared library using virtual memory.

  - System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Actual pages where the libraries reside in physical memory are shared by all the processes (see Fig. 3).
  - Similarly, virtual memory enables processes to share memory. Two or more processes can communicate through the use of shared memory (see Fig. 3).
  - Virtual memory can allow pages to be shared during process creation with the $fork()$ system call, thus speeding up process creation.

1.2 Demand Paging

- Consider how an executable program might be loaded from disk into memory.
  - One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory.
  - An alternative strategy is to initially load pages only as they are needed. This technique is known as demand paging and is commonly used in virtual memory systems.
- A demand-paging system is similar to a paging system with swapping (see Fig. 4) where processes reside in secondary memory (usually a disk).

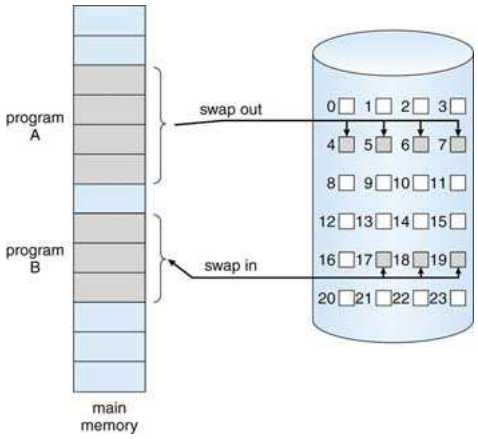

Figure 4: Transfer of a paged memory to contiguous disk space.

- When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed.
- A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use **pager**, rather than swapper, in connection with demand paging.

1.2.1 Basic Concepts

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- It avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
- Some form of hardware support is needed to distinguish between the pages that are in memory and the pages that are on the disk.
- The valid -invalid bit scheme can be used for this purpose.
  - This time however, when this bit is set to "valid", the associated page is both legal and in memory.
  - If the bit is set to "invalid", the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.
  - The page-table entry for a page that is brought into memory is set as usual,
  - but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk (see Fig. 5).

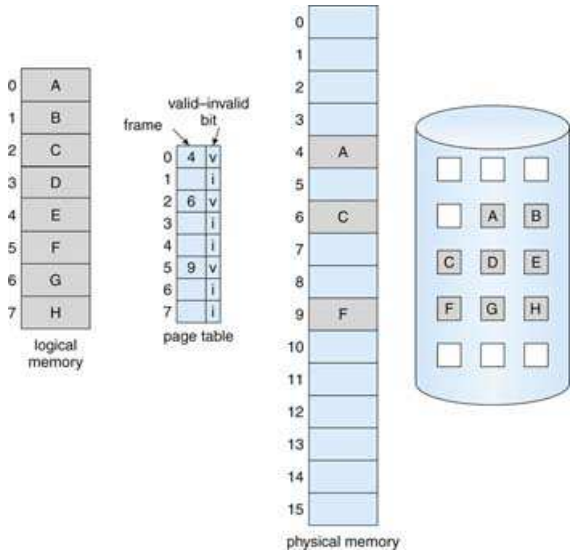

Figure 5: Page table when some pages are not in main memory.

- While the process executes and accesses pages that are memory resident, execution proceeds normally.
- But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page-fault trap**.
- The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the OS. The procedure for handling this page fault is straightforward (see Fig. 6).

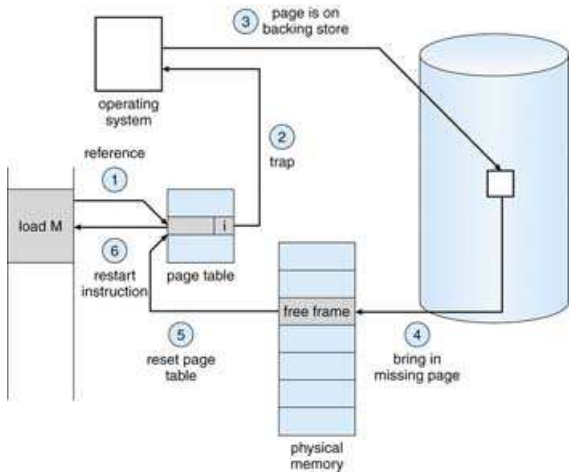

Figure 6: Steps in handling a page fault.

1. We check an internal table (in PCB) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table.
6. We restart the instruction that was interrupted by the trap.

- In the extreme case, we can start executing a process with no pages in memory.
  - When the OS sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page.
  - After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory.
- At that point, it can execute with no more faults. This scheme is **pure demand paging**: *Never bring a page into memory until it is required.*
- Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction.
- This situation would result in unacceptable system performance (but fortunately this behavior is exceedingly unlikely).
- Programs tend to have **locality of reference** which results in reasonable performance from demand paging.
- Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in exactly the same place and state.
  - If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again.
  - If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

1.2.2 Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system. Let's compute the **effective access time** for a demand-paged memory.
  - For most computer systems, the memory-access time, denoted ma, ranges from 10 to 200 nanoseconds.
  - As long as we have no page faults, the effective access time is equal to the memory access time.

- If, however a page fault occurs, we must first read the relevant page from disk and then access the desired word.
- Let p be the probability of a page fault ($0 \leq p \leq 1$). We would expect p to be close to zero -that is, we would expect to have only a few page faults.
- The effective access time is then

```
effective access time = (1 - p)*ma + p*page fault time
```

- We are faced with three major components of the page-fault service time:
  1. Service the page-fault interrupt.
  2. Read in the page.
  3. Restart the process.
- The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each.
- The page-switch time, however, will probably be close to 8 milliseconds.
  -
  - A typical hard disk has an average latency of 3 milliseconds, a seek of 5 milliseconds, and a transfer time of 0.05 milliseconds.
- Thus, the total paging time is about 8 milliseconds, including hardware and software time.
- If we take an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, then the effective access time in nanoseconds is

```
effective access time = (1 - p)*(200) + p*(8 milliseconds)
                      = (1 - p)*200 + p*8,000,000
      = 200 + 7,999,800 x p.
```

- We see, then, that the effective access time is directly proportional to the **page-fault rate**.
- If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. The computer will be slowed down by a factor of 40 because of demand paging!

- It is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.
- An additional aspect of demand paging is the handling and overall use of swap space.
  - Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used.
  - The system can therefore gain better paging throughput by copying an entire file image into the swap space at process startup and then performing demand paging from the swap space.
  - Another option is to demand pages from the file system initially but to write the pages to swap space as they are replaced.

1.3 Copy-on-Write

- Process creation using the $fork()$ system call may initially bypass the need for demand paging by using a technique similar to page sharing.
- This technique provides for rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.
- Recall that the $fork()$ system call creates a child process as a duplicate of its parent.
  - Traditionally, $fork()$ worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent.
  - However, considering that many child processes invoke the $exec()$ system call immediately after creation, the copying of the parent's address space may be unnecessary.
- Alternatively, we can use a technique known as **copy-an-write**, which works by allowing the parent and child processes initially to share the same pages.
- These shared pages are marked as copy-an-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created (see Figs. 7 and 8).

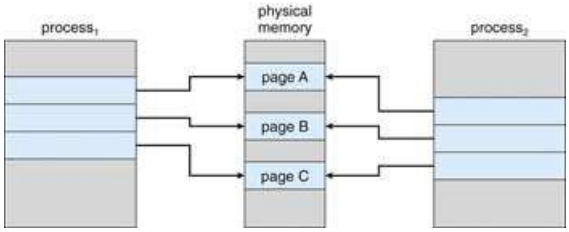

Figure 7: Before process 1 modifies page C.

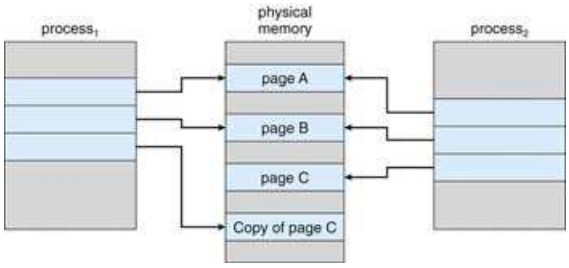

Figure 8: After process 1 modifies page C.

- Only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child.
- Copy-on-write is a common technique used by several OSs, including Windows XP, Linux, and Solaris.

1.4 Page Replacement

- If we increase our degree of multiprogramming, we are over-allocating memory.
  - If we run six processes, each of which is ten pages in size but actually uses only five pages, we have higher CPU utilization and throughput, with ten frames to spare.
  - It is possible that each of these processes may suddenly try to use all ten of its pages resulting in a need for sixty frames.
- Further, consider that system memory is not used only for holding program pages. Buffers for I/O also consume a significant amount

of memory. This use can increase the strain on memory-placement algorithms.

- Over-allocation of memory manifests itself as follows (see Fig. 9).
  - While a user process is executing, a page fault occurs.
  - The OS determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use.

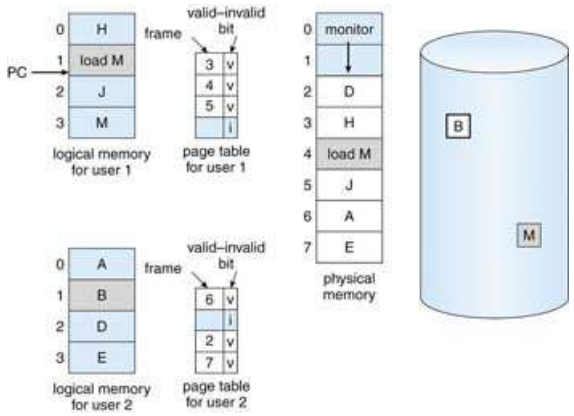

Figure 9: Need for page replacement.

- The OS could swap out a process, freeing all its frames and reducing the level of multiprogramming.

1.4.1 Basic Page Replacement

- Page replacement takes the following approach (see Fig. 10).
  - If no frame is free, we find one that is not currently being used and free it.
  - We can free a frame by writing its contents to swap space and changing the page table (and al other tables) to indicate that the page is no longer in memory.
  - We can now use the freed frame to hold the page for which the process faulted.

- We modify the page-fault service routine to include page replacement:

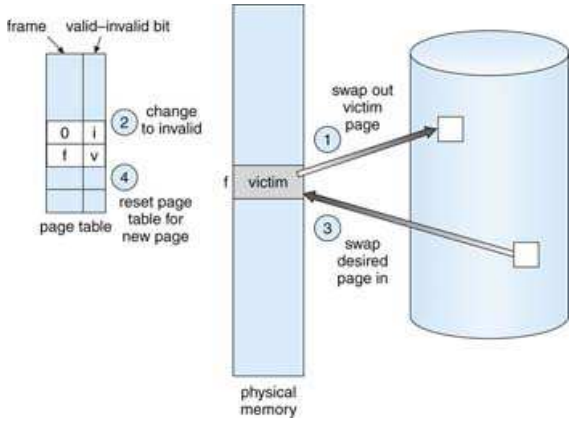

Figure 10: Page replacement.

1. Find the location of the desired page on the disk.
2. Find a free frame:
   a If there is a free frame, use it.
   b If there is no free frame, use a page-replacement algorithm to select a victim frame.
   c Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

- Notice that, if no frames are free, two page transfers (one out and one in) are required.
- We can reduce this overhead by using a **modify bit** (or **dirty bit**).
- The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.
- When we select a page for replacement, we examine its modify bit.
  – If the bit is set, we know that the page has been modified since it was read in from the disk (write that page to the disk).
  – If the modify bit is not set, the page has not been modified since it was read into memory (not write the memory page to the disk: It is already there).

- This technique also applies to read-only pages (for example, pages of binary code).
- This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.
- Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory.
- We must solve two major problems to implement demand paging:
    1. develop a **frame-allocation algorithm**. If we have multiple processes in memory, we must decide how many frames to allocate to each process.
    2. develop a **page-replacement algorithm**. When page replacement is required, we must select the frames that are to be replaced.
- Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.
- Second major problem will be discussed firstly.
- For a given page size (and the page size is generally fixed by the hardware or system), we need to *consider only the page number, rather than the entire address.*
- If we have a reference to a page p, then any immediately following references to page p will never cause a page fault (page p will be in memory after the first reference).
- For example, if we trace a particular process, we might record the following address sequence:

  `0100,0432,0101,0612,0102,0103,0104,0101,0611,0102,0103,`
  `0104,0101,0610,0102,0103,0104,0101,0609,0102,0105`

- At 100 bytes per page, this sequence is reduced to the following reference string:

  `1,4,1,6,1,6,1,6,1,6,1`

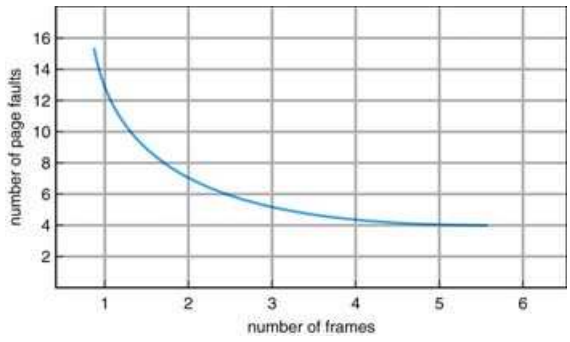

Figure 11: Graph of page faults versus number of frames.

- As the number of frames increases, the number of page faults drops to some minimal level (see Fig. 11).
- Next several page-replacement algorithms are illustrated. The following reference string will be used to exemplify for a memory with three frames.

```
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
```

1.4.2 FIFO Page Replacement

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- A FIFO replacement algorithm *associates with each page the time when that page was brought into memory.*
- When a page must be replaced, the **oldest page is chosen**.
- For our example reference string, our three frames are initially empty.

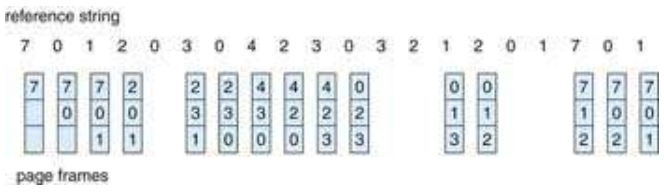

Figure 12: FIFO page-replacement algorithm.

  - The first three references (7, 0, 1) cause page faults and are brought into these empty frames.
  - The next reference (2) replaces page 7, because page 7 was brought in first.
  - Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
  - The first reference to 3 results in replacement of page 0, since it is now first in line.
  - Because of this replacement, the next reference, to 0, will fault.
  - Page 1 is then replaced by page O. This process continues and there are 15 faults altogether.

- The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.
- On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed.
- On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.
- Notice that, even if we select for replacement a page that is in active use, everything still works correctly.
  - After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page.
  - Some other page will need to be replaced to bring the active page back into memory.
  - Thus, a bad replacement choice increases the page-fault rate and slows process execution.
  - It does not cause incorrect execution.

1.4.3 Optimal Page Replacement

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms (called OPT or MIN). It is simply this:

```
Replace the page that will not be used
for the longest period of time.
```

- Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.
- For example, on our sample reference string, the **optimal page-replacement algorithm** would yield nine page faults (see Fig. 13).

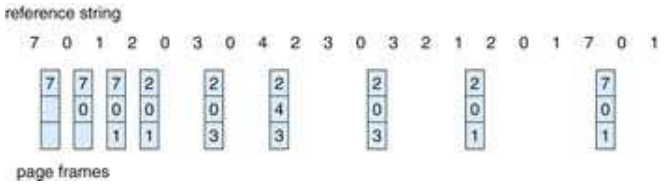

Figure 13: Optimal page-replacement algorithm.

  - The first three references cause faults that fill the three empty frames.
  - The reference to page 2 replaces page 7, because 7 will not be used until reference 18,
  - whereas page 0 will be used at 5, and page 1 at 14.
  - The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.
- With only nine page faults, optimal replacement is much better than a FIFO algorithm, which resulted in fifteen faults.
- If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string (similar situation with the SJF CPU-scheduling algorithm).
- As a result, the optimal algorithm is used mainly for comparison studies.

1.4.4 LRU Page Replacement

- The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that
  - the FIFO algorithm uses the time when a page was brought into memory,

  - whereas the OPT algorithm uses the time when a page is to be used.

- If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time (see Fig. 14).

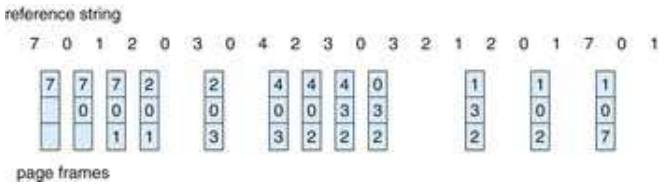

Figure 14: LRU page-replacement algorithm.

- This approach is the **least-recently-used (LRU) algorithm**. The result of applying LRU replacement to our example reference string is shown in Fig. 14. The LRU algorithm produces 12 faults.
  - Notice that the first 5 faults are the same as those for optimal replacement.
  - When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently.
  - Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
  - When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.

- Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

1.5 Allocation of Frames

- Now, first major problem mentioned in Section 1.4.1 will be discussed. How do we allocate the fixed amount of free memory among the various processes?

- If we have 93 free frames and two processes, how many frames does each process get?

- The simplest case is the single-user system.
  - Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system has 128 frames.
  - The OS may take 35 K8, leaving 93 frames for the user process.
- Under pure demand paging, all 93 frames would initially be put on the free-frame list.
  - When a user process started execution, it would generate a sequence of page faults.
  - The first 93 page faults would all get free frames from the free-frame list.
  - When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on.
  - When the process terminated, the 93 frames would once again be placed on the free-frame list.
- There are many variations on this simple strategy. We can require that the OS allocate all its buffer and table space from the free-frame list.
- When this space is not in use by the OS, it can be used to support user paging. The user process is allocated any free frame.

1.5.1 Allocation Algorithms

- The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames. This scheme is called **equal allocation**.
  - For instance, if there are 93 frames and five processes, each process will get 18 frames.
  - The leftover three frames can be used as a free-frame buffer pool.
- An alternative is to recognize that various processes will need differing amounts of memory.
  - Consider a system with a 1-KB frame size.
  - If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames.

  - The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.

- To solve this problem, we allocate available memory to each process according to its size (proportional allocation).

- Let the size of the virtual memory for process p_i be s_i, and define

$$S = \sum s_i$$

- Then, if the total number of available frames is m, we allocate a_i frames to process p_i, where a_i is approximately

$$a_i = \frac{s_i}{S * m}$$

- For proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

```
10/137 x 62 ~ 4,
127/137 x 62 ~ 57.
```

- In this way, both processes share the available frames according to their "needs", rather than equally.

- In both equal and proportional allocation, of course, the allocation may vary according to the multiprogramming level.

  - If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process.
  - If the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

1.5.2 Global versus Local Allocation

- Another important factor in the way frames are allocated to the various processes is page replacement.

- With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories:

1. **Global replacement**. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.
2. **Local replacement**. Local replacement requires that each process select from only its own set of allocated frames.

- With a local replacement strategy, the number of frames allocated to a process does not change.
- With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose its frames for replacement).
- Global replacement generally results in greater system throughput and is therefore the more common method.

1.6 Thrashing

- If the number of frames allocated to a low-priority process falls below the minimum number required, we must suspend that process's execution. We should then page out its remaining pages, freeing all its allocated frames.
- In fact, look at any process that does not have "enough" frames.
  - If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.
  - At this point, it must replace some page.
  - However, since all its pages are in active use, it must replace a page that will be needed again right away.
  - Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
- This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing.

1.6.1 Cause of Thrashing

- Thrashing results in severe performance problems. Consider the following scenario (see Fig. 15);

- The OS monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.

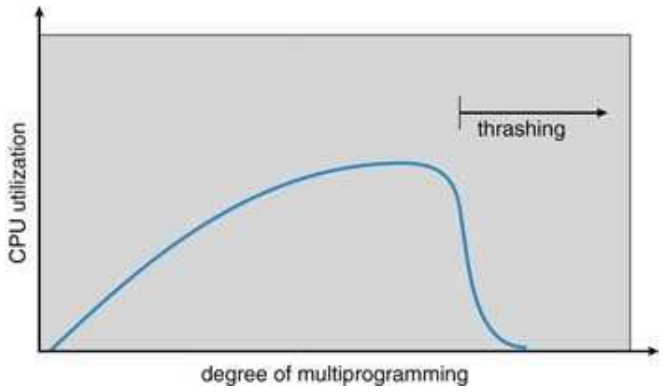

Figure 15: Thrashing.

  - Now suppose that a process enters a new phase in its execution and needs more frames.
  - It starts faulting and taking frames away from other processes (global page-replacement algorithm).
  - These processes need those pages, however, and so they also fault, taking frames from other processes.
  - These faulting processes must use the paging device to swap pages in and out.
  - As processes wait for the paging device, CPU utilization decreases.

- The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result.

  - The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device.
  - As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.

- Thrashing has occurred. The page-fault rate increases tremendously. As a result, the effective memory-access time increases.

- No work is getting done, because the processes are spending all their time paging.
- As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached.
- If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.
- At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.